Università degli Studi di Parma
Dipartimento di Ingegneria e Architettura
Sistemi operativi e in tempo reale – a.a. 2022/23

Costrutti linguistici per la programmazione concorrente in ambiente globale *Regioni Critiche*

prof. Stefano Caselli
stefano.caselli@unipr.it

Esempi di uso dei semafori

- mutua esclusione:

```
wait(mutex)                    (v.i.  mutex=1)
 <sez. crit.>
signal(mutex)
```

- scambio di messaggi (produttore - consumatore):

```
      P                          C
wait(s1)                   wait(s2)      (v.i. s1=N, s2=0)
 ....                        ....
signal(s2)                 signal(s1)
```

Esempi di uso dei semafori

- semaforo privato:

acquisizione

```
wait(mutex);
  ....
if condition then signal(s_i);
signal(mutex);
wait(s_i);
```

rilascio

```
wait(mutex);
   ....
condition := true;
signal(s_k);
   ....
signal(mutex);
```

(v.i. mutex=1, s_i=0)

Possibili errori nell'uso dei semafori

P1, P2

```
wait(mutex);
 …
signal(mutex);
```

P3

```
signal(mutex);
 …
wait(mutex);
```

- Se P3 è in sezione critica e viene interrotto da P1 o P2? Possibili errori time-dependent

P1

```
wait(mutex);
<sez crit1>
wait(mutex);
```

P2

```
wait(mutex);
<sez crit 2>
signal(mutex);
```

- Se P1 entra in questo ramo di codice, ci sarà deadlock per P1 e P2

Uso dei semafori: considerazioni

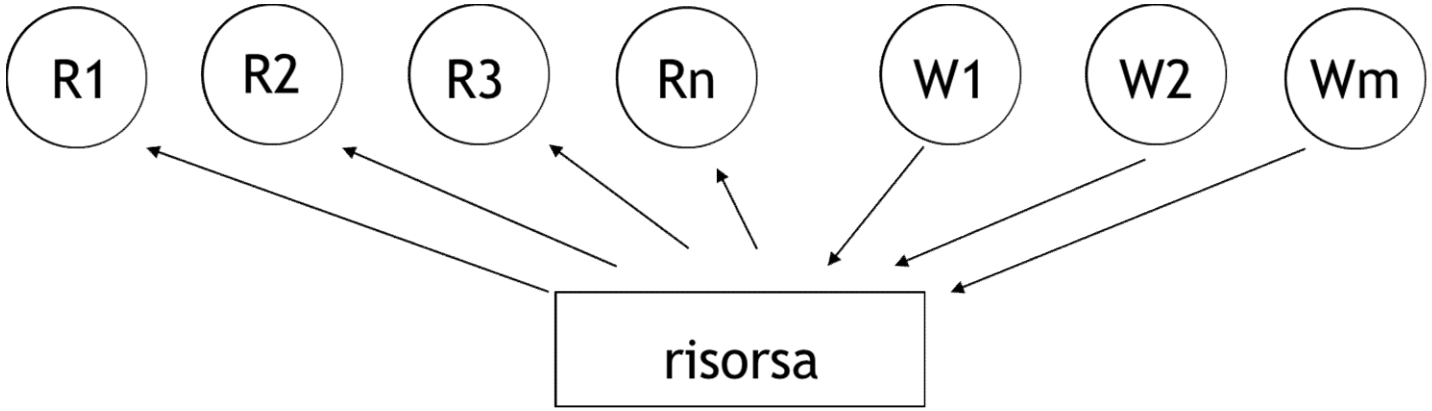
- wait(s) e signal(s) possono risolvere *qualsiasi problema* di sincronizzazione, quindi non è possibile rilevare un loro uso scorretto in fase di compilazione
- Problemi di sincronizzazione complicati richiedono più semafori → più possibilità di commettere errori!
- Per superare gli inconvenienti legati all'uso dei semafori, nella programmazione concorrente si è cercato di spostare più *controlli a tempo di compilazione*
- → *Costrutti linguistici* di più alto livello, tradotti dal compilatore in termini di wait(s) e signal(s), per incrementare sicurezza ed affidabilità
- Questi costrutti linguistici si sono nel tempo trasformati in *pattern*

Uso dei semafori: considerazioni

- I semafori sono uno strumento «dual purpose»: mutua esclusione, segnalazione di eventi
- Nella pratica si è osservato che questa caratteristica li rende complicati da usare e comprendere
- --> meglio mascherare o adottare *strumenti distinti* per le due funzioni

Problema dei Lettori-Scrittori

- In letteratura: *Readers and Writers*



- I thread Lettori possono usare la risorsa anche contemporaneamente; la lettura è *non consumativa*
- I thread Scrittori devono avere accesso esclusivo alla risorsa, perché la modificano

Problema dei Lettori-Scrittori

- Soluzione 1 «*Reader preference*»:
  - Un thread lettore attende solo se la risorsa è già stata assegnata ad un thread scrittore. Nessun lettore aspetta perché c'è uno scrittore in attesa
  - Rischio di starvation per i thread scrittori

- Soluzione 2 «*Writer preference*»:
  - Un thread lettore attende se c'è un thread scrittore in attesa
  - Rischio di starvation per i thread lettori

Readers and Writers - Soluzione 1

var readercount: *integer* (v.i. = 0)
mutex, w: *semaphore* (v.i. = 1)

Reader	**Writer**
wait(mutex);	
readercount := readercount + 1;	
if readercount = 1 *then* *wait*(w);	
signal(mutex);	*wait*(w);
<lettura>	<scrittura>
wait(mutex);	*signal*(w);
readercount := readercount - 1;	
if readercount = 0 *then* *signal*(w);	
signal(mutex);	

Readers and Writers - Soluzione 1

- Un thread lettore con *wait(w)* impedisce l'accesso ai thread scrittori, oppure si blocca perché c'è una scrittura in corso. In questo caso gli altri lettori rimangono bloccati su mutex
- E' una soluzione *weak reader preference*: quando c'è un thread scrittore sulla risorsa, eventuali ulteriori scrittori lo potranno seguire in ordine FIFO fino al primo lettore (ordinamento della coda su *w*)
- Gli altri lettori si potranno aggiungere quando il primo lettore accede alla risorsa

Readers and Writers - Soluzione 2

var readercount, writercount: *integer* (v.i. = 0)
mutex1, mutex2, mutex3, w, r: *semaphore* (v.i. = 1)

Reader

```
wait (mutex3);
wait (r);
  wait (mutex1);
  readercount := readercount + 1;
  if readercount = 1 then wait (w);
  signal (mutex1);
signal (r);
signal (mutex3);
<lettura>
```

// segue Reader

```
wait (mutex1);
readercount := readercount - 1;
if readercount = 0 then signal (w);
signal (mutex1);
```

Readers and Writers - Soluzione 2

Writer

```
wait (mutex2);
writercount := writercount + 1;
if  writercount = 1 then wait (r);
signal (mutex2);
 wait (w);
 <scrittura>
 signal (w);
wait (mutex2);
writercount := writercount - 1;
if writercount = 0 then signal (r);
signal (mutex2);
```

- Soluzione *strong writer preference*

Readers and Writers – Discussione soluzione 2

- La soluzione *writer preference* precedente richiede 5 (o 4) semafori: complicata!
- Ruolo del semaforo mutex3 nei lettori:
  - Ha un impatto sull'efficienza e il rigore della politica *writer preference*, non sulla correttezza
  - Quando c'è uno scrittore in sezione critica c'è un solo lettore sul semaforo *r*, gli altri sono su *mutex3*; eventuali scrittori sono fermi su *wait(w)*
  - Situazione di interesse: quando l'ultimo scrittore lascia la risorsa ed esegue *signal(r)*: senza *mutex3* tutti i lettori avanzerebbero sulla risorsa. Un eventuale scrittore che arrivi immediatamente in questa fase dovrebbe accodarsi a tutti con *wait(r)*
  - Con la presenza di *mutex3* lo scrittore può precedere quei lettori che non hanno completato il protocollo di accesso
  - E' quindi un rafforzamento della politica *writer priority*

Problema dei Lettori-Scrittori

- Importante paradigma di interazione tra processi e thread nelle applicazioni reali
- La identificazione di thread che operano come "lettori" consente una *ottimizzazione dei tempi di accesso rispetto alla serializzazione generale* degli accessi alla risorsa
- *Classificazione principale* in base al comportamento del lettore in arrivo:
  - se il lettore entra comunque quando ci sono già lettori sulla risorsa → *reader preference*
  - se il lettore attende in caso di presenza di scrittore in coda → *writer preference*

Problema dei Lettori-Scrittori

- *Classificazione secondaria* in base al comportamento del protocollo quando lo scrittore rilascia la risorsa e sono in attesa thread di entrambi i tipi:
  - se viene comunque scelto un lettore → *strong reader preference*, oppure *weak-weak writer preference*
  - se viene comunque scelto uno scrittore → *weak-weak reader preference*, oppure *strong writer preference*
  - se la scelta è FIFO o non deterministica → *weak reader preference*, oppure *weak writer preference*

Problema dei Lettori-Scrittori

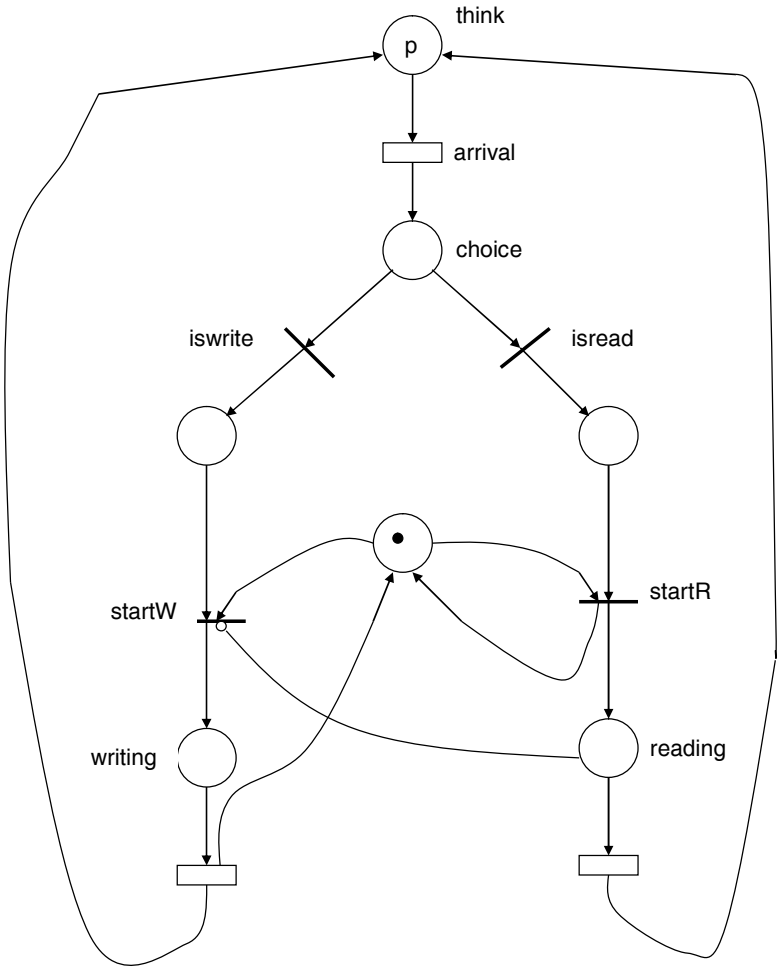
- Le diverse politiche si differenziano per la possibilità di starvation per uno o entrambi i tipi di thread e per il grado di concorrenza. La politica più appropriata dipende dal mix dei thread. Sono possibili analisi mediante strumenti di modellazione e di performance evaluation
- Quali politiche, in dettaglio, sono realizzate dalle due soluzioni basate su semafori viste?

Problema dei Lettori-Scrittori

- All'uscita dello scrittore viene comunque scelto un lettore:
  - *strong reader preference* → possibile starvation W
  - *weak-weak writer preference* → no starvation, ma bassa concorrenza R
- All'uscita dello scrittore viene comunque scelto uno scrittore:
  - *weak-weak reader preference* → possibile starvation R e W
  - *strong writer preference* → possibile starvation R
- Scelta FIFO o non deterministica:
  - *weak reader preference* → possibile starvation W
  - *weak writer preference* → no starvation ?

- Quale politica scegliete?

Modello a rete di Petri



Riassunto

- Semaphores are overly complicated
- Use sparingly, only when needed

- Let's try to do better

- Un'idea ricorrente (up and down): spostare parte dei controlli a tempo di compilazione
  - → supporti per la concorrenza a livello linguistico (*Regioni Critiche Condizionali* e *Monitor*)
  - se non supportati dal linguaggio, sono *pattern* di uso dei meccanismi di base del SO ai fini della concorrenza

Regioni Critiche

- Regione critica semplice (Brinch Hansen, Hoare – 1972-75)
- Dichiarazione: `var v: shared T`
- Uso della regione critica:

  `region v do S1; S2; … Sn end`

- La sequenza <S1; S2; … Sn> costituisce una sezione critica. Gli statement hanno accesso alla variabile *shared v*
- Il *compilatore* può verificare che la variabile *v* sia usata esclusivamente entro la regione critica e può realizzare correttamente alla mutua esclusione
- La regione critica semplice consente di risolvere (solo) problemi di mutua esclusione

Regione Critica: Esempio

```
var  pila: shared  record
                        top: 0 .. N
                        stack: array  [0 .. N-1] of  messaggio
                   end
begin   top := 0  end      /* valore iniziale */

procedure  inserimento (y: messaggio);
  region  pila do
        if  top = N  then  /* pila piena */
        else begin
                stack[top] := y;
                top := top + 1
             end
  end
end
```

Regione Critica: Esempio

```
procedure  prelievo (var  x: messaggio);
   region  pila do
                    if  top = 0  then  /* pila vuota */
                    else begin
                              top := top - 1;
                              x := stack[top]
                         end
   end
end
```

- La mancata gestione dei casi «pila piena» e «pila vuota» evidenzia come il costrutto RC semplice consenta di realizzare solo la mutua esclusione e non l'esecuzione condizionata

Regione Critica: realizzazione

- Il compilatore realizza il costrutto Regione Critica (RC) come segue:

- Dichiarazione della RC:

  var v: *shared* T; → mutex v.i. = 1

- Istruzione in cui si usa la RC:

  region v *do* S *end*; → *wait*(mutex)
  <S>
  signal(mutex)

Regione Critica: proprietà

- Il compilatore controlla che i processi accedano a *v* solo entro la regione critica
- Il compilatore può riconoscere situazioni di *deadlock potenziale*:

var v1, v2: *shared* T;

P1

…

region v1 *do*

region v2 *do* ... *end*;

end;

P2

…

region v2 *do*

region v1 *do* ... *end*;

end;

Regione Critica Condizionale

- La dichiarazione della *Regione Critica Condizionale (RCC)* è identica a quella della RC semplice:

```
var v: shared T
```

- Uso della regione critica condizionale:

```
region v when B do S1; S2; … Sn end
```

- La clausola `when` consente di specificare quando può essere eseguita la sequenza *regione critica* < `S1; S2; … Sn` >
- Consente di ritardare il completamento di una regione critica fino a quando non si verifica la *condizione* B

Regione Critica Condizionale

- Quando il thread entra nella RC, viene valutata la *condizione* B (boolean expression): se B è vera la RC è completata *eseguendo* S1, S2, Sn; diversamente il thread *libera la sezione critica* e *si sospende* in una coda associata alla variabile *v*
- Nella espressione booleana B compaiono in genere elementi della struttura dati *v* (il dato condiviso protetto dal costrutto). Pertanto una successiva manipolazione di *v* da parte di un thread che ha potuto accedere (avendo trovato *la propria condizione* verificata) potrà rendere vera la condizione B di uno o più thread sospesi
- *Ogni volta che un thread esce dalla RC tutti i thread* eventualmente sospesi vengono *risvegliati* e rivalutano *la propria condizione* di accesso

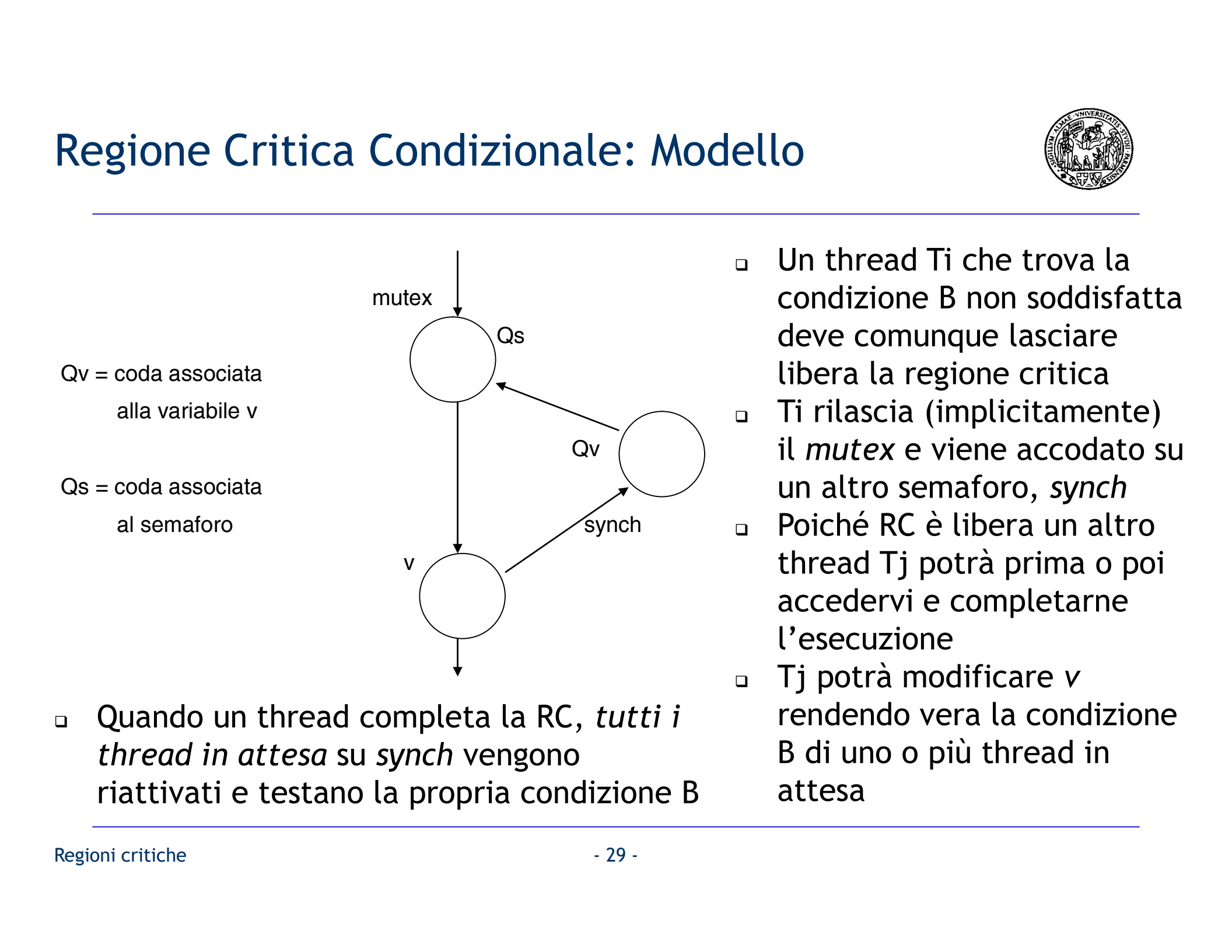
Regione Critica Condizionale: Scambio di messaggi

```
var  mailbox: shared  record
                                buffer: array [0 .. N-1] of  char;
                                testa, coda: 0 .. N-1;
                                cont: 0 .. N;     /* celle occupate */
                            end
v.i.  testa := coda := cont := 0;
procedure  send (x: char);
        begin
                region  mailbox  when  cont < N  do
                                buffer[coda] := x;
                                coda := (coda + 1) mod N;
                                cont := cont + 1;
                end;
        end;
```

Regione Critica Condizionale: Scambio di messaggi

```
procedure  receive (var  x: char);
        begin
                region  mailbox  when  cont > 0  do
                                x := buffer[testa];
                                testa := (testa + 1) mod N;
                                cont := cont - 1;
                end;
        end;
```

Regione Critica Condizionale: Modello



Qv = coda associata alla variabile v

Qs = coda associata al semaforo

- Quando un thread completa la RC, *tutti i thread in attesa* su *synch* vengono riattivati e testano la propria condizione B

- Un thread Ti che trova la condizione B non soddisfatta deve comunque lasciare libera la regione critica
- Ti rilascia (implicitamente) il *mutex* e viene accodato su un altro semaforo, *synch*
- Poiché RC è libera un altro thread Tj potrà prima o poi accedervi e completarne l'esecuzione
- Tj potrà modificare *v* rendendo vera la condizione B di uno o più thread in attesa

Realizzazione della Regione Critica Condizionale

```
var v: shared T;
```

=> il compilatore istanzia le seguenti variabili di supporto:

```
var     mutex: semaphore  initial  (1);
        synch: semaphore  initial  (0);
        cont:  integer  initial  (0);
```

Realizzazione della Regione Critica Condizionale

```
region v when B do S end;
=>
```

```
wait (mutex);
 while  not  B do
 begin
    cont := cont + 1;
    signal (mutex);   // libera sez. crit.
    wait (synch);     // sosp. su coda Qv
    wait (mutex);
 end
 <S>     // accesso a sez. crit. utente
```

```
// segue
while  cont <> 0 do
 begin // risveglia tutti i thread
       // per la rivalutazione
       // delle condizioni
    signal (synch);
    cont := cont - 1;
 end
signal (mutex);
```

Regioni Critiche: discussione

- Vantaggi ottenibili con le regioni critiche semplici e condizionali:
  - a) *maggiore chiarezza* nel programma
  - b) *controlli a tempo di compilazione*

- Il compilatore:
  - può controllare che l'accesso a variabili *shared* avvenga solo entro le regioni critiche;
  - provvede direttamente ad assicurare la mutua esclusione, evitando così l'uso dei semafori da parte del programmatore

Regioni Critiche: discussione

- Problemi:
- Il programmatore *non controlla l'ordine* con cui i thread hanno accesso alla risorsa comune:
  - *Tutti* i thread sospesi in Qv vengono trasferiti in Qs, ed *il primo* per il quale è vera la condizione di sincronizzazione ha accesso alla risorsa
  - Non è possibile affidare il controllo ad un particolare thread. Non è possibile imporre una specifica politica di scheduling, qualora necessario
- Sussistono problemi di *starvation*, risolubili dando *priorità ai thread della coda Qv* (synch) rispetto a quelli già in Qs (mutex)

Regioni Critiche: discussione

- La regione critica condizionale, così come definita, consente di operare sincronizzazioni solo all'inizio della sezione critica
- Per superare questa limitazione il costrutto è stato modificato con la clausola *await* come segue:

```
region v do begin
            S1;
            await (B);
            S2;
        end;
```

Regioni Critiche: discussione

- In presenza di più thread o processi nella coda Qv, *tutti* vengono risvegliati e provvedono a valutare *la propria condizione B*, con eventuale successiva nuova sospensione
  - → La soluzione può determinare un *numero elevato di cambi di contesto*
- ➔ le regioni critiche condizionali sono adatte a sistemi con poche interazioni tra i processi *(loosely connected processes)*

Realizzazione alternativa della Regione Critica Condizionale

- `"var v shared T;"`

```
var mutex semaphore initial  1;
    synch1,synch2 semaphore initial 0;
    cont1,cont2 integer initial  0;
```

- `"region v when B do S end;"`

- (segue)

```
wait(mutex);
while not B do {
        cont1 := cont1 + 1;
        if cont2 > 0 signal(synch2)
        else signal(mutex);
        wait(synch1);
        cont1 := cont1 - 1;
        cont2 := cont2 + 1;
        if cont1 > 0 signal(synch1)
        else signal(synch2);
        wait(synch2);
        cont2 := cont2 - 1; }
        <S>
```

Realizzazione alternativa della Regione Critica Condizionale

- Rilascio della Regione Critica

- Segue - epilogo di "`region v when B do S end;`":

…
<S>
if cont1 > 0 *signal* (synch1)
else if cont2 > 0 *signal* (synch2)
else signal (mutex);

Realizzazione alternativa della Regione Critica Condizionale

- Si supponga che i primi n processi trovino la propria condizione *Bi non soddisfatta*: → n processi in coda sul semaforo synch1
- Il successivo processo Pj trova *Bj soddisfatta*; dopo avere completato S fa avanzare il primo processo da synch1 a synch2
- Si apre una fase di *risveglio a catena*, a seguito della quale tutti i processi precedentemente su synch1 passano su synch2
- L'*ultimo* processo della catena risveglia il primo processo in coda su synch2 prima di sospendersi su synch2 a sua volta; nel caso ci sia un solo processo in circolo, fa una signal a proprio favore rendendo passante la successiva wait

Realizzazione alternativa della Regione Critica Condizionale

- Un processo Pi che ha superato synch2 può verificare la condizione B: se trova B soddisfatta, esegue S e quindi abiliterà il successivo processo presente su synch2 o, se non ci sono processi su synch2, sul mutex
- Se trova B non soddisfatta, Pi si sospende su synch1 e risveglia un processo su synch2, se presente, o riabilita il mutex
- Prima di abilitare il mutex, tutti i processi bloccati all'interno hanno la possibilità di testare una volta la propria condizione B per ogni processo che esegue con successo la regione
- In definitiva: ogni processo in attesa entro la sezione critica verifica nuovamente la condizione ed eventualmente accede prima che sia abilitato un processo esterno su mutex

Esempi di applicazione della RCC

- Problema Lettori-Scrittori

Lettori-Scrittori: Soluzione mediante Regione Critica Condizionale

```
var   rw_buff: shared  record
                       num_lettori:    integer      initial  0;
                       num_scrittori:  integer      initial  0;
                       occupato:       boolean      initial  false;
                       end
```

- Definiamo quattro procedure di accesso alle variabili di rw_buff: Inizio_lettura(), Fine_lettura(), Inizio_scrittura(), Fine_scrittura()

Lettori-Scrittori: Soluzione mediante Regione Critica Condizionale

- L'accesso vero e proprio al buffer condiviso non avviene entro la Regione Critica. Perché?

```
        R                          W
        …                          …
Inizio_lettura()          Inizio_scrittura()
<Read>                    <Write>
Fine_lettura()            Fine_scrittura()
        …
```

Lettori-Scrittori: Soluzione mediante Regione Critica Condizionale

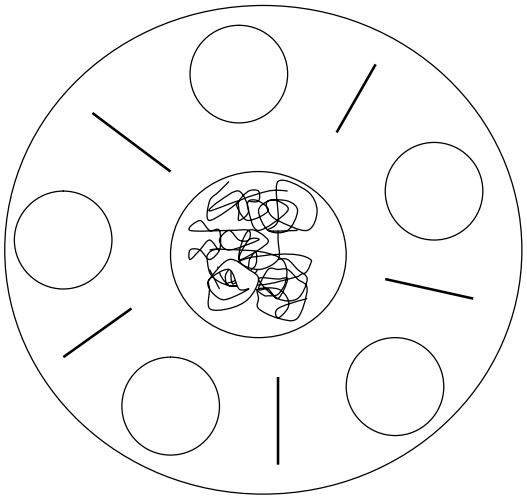
```
procedure  Inizio_lettura();
        begin
                region  rw_buff do begin
                                        await  (num_scrittori = 0);
                                        num_lettori := num_lettori + 1;
                                    end
        end
procedure  Fine_lettura();
        begin
                region  rw_buff do begin
                                        num_lettori := num_lettori - 1;
                                    end
        end
```

Lettori-Scrittori: Soluzione mediante Regione Critica Condizionale

```
procedure  Inizio_scrittura();
  begin
        region rw_buff do begin
                                    num_scrittori := num_scrittori + 1;
                                    await ((not occupato) and (num_lettori = 0));
                                    occupato := true;
                               end
  end
procedure  Fine_scrittura();
   begin
        region rw_buff do  begin
                                    num_scrittori := num_scrittori - 1;
                                    occupato := false;
                               end
        end
```

Filosofi a cena: Soluzione mediante Regione Critica Condizionale

- N filosofi dal robusto appetito, che alternano pensiero e cibo; una grande teglia di spaghetti al centro
- N chopstick: per cibarsi i filosofi devono utilizzare due chopstick (?? – cucina olandese)
- I chopstick devono essere utilizzati in modo esclusivo



Filosofi a cena: Soluzione mediante Regione Critica Condizionale

```
var  philosophers: shared  record
        state: array  [0 .. N - 1]  of  (thinking, eating) initial  thinking;
     end                                  // stato "hungry" espresso implicitamente
procedure  pickup (i:  0 .. N-1);  // eseguita dall'i-esimo Filosofo per mangiare
  begin
        region  philosophers when (state [(i+N-1) mod N] <> eating
                                          and  state [(i+1) mod N] <> eating)
        do  state [i] := eating;
  end
procedure  putdown (i:  0 .. N-1);
  begin
        region  philosophers  when  true  do  state [i] := thinking;
  end
```

Filosofi a cena: Soluzione mediante Regione Critica Condizionale

- Il filosofo che esegue la procedura *putdown* riattiva tutti i filosofi sospesi, che verificano nuovamente la condizione
- La condizione potrà eventualmente risultare soddisfatta per uno o entrambi i filosofi adiacenti, oppure per nessuno
- Se entrambi i filosofi adiacenti sono in attesa e trovano la propria condizione vera, si serializzeranno solo nell'accesso alle variabili di controllo e potranno mangiare in parallelo